FICHA 4
ALGORITMIA E ALGORITMOS DE ORDENAÇÃO

4.1 Objetivos

Objetivos que o aluno é suposto atingir com esta ficha:

- Entender o funcionamento de dois algoritmos simples de ordenação, o *selection sort* e o *bubble sort*;
- Saber implementar estes dois algoritmos em C/C++ para ordenar uma tabela de inteiros, uma tabela de *strings* e uma tabela de *structs*, tanto por ordem crescente, como por ordem decrescente;
- Saber utilizar a função `qsort()` disponível no ANSI C para ordenar uma tabela de dados;
- Saber ordenar um ficheiro de dados desordenados, lendo a totalidade do ficheiro para memória, ordenando os dados em memória e finalmente escrevendo os dados ordenados de volta para o ficheiro.
- Saber calcular a complexidade de um algoritmo, expressando-a na notação O-grande.

4.2 Ordenação

A ordenação de informação sempre foi uma necessidade nas mais variadas aplicações:

- Ordenação de livros em bibliotecas, por título, por autor ou por outros campos;
- Ordenação dos contribuintes de uma determinada repartição de finanças;
- Ordenação de nomes numa lista telefónica ou numa simples agenda telefónica pessoal;
- Ordenação de uma lista de clientes de uma empresa;
- Ordenação de peças contidas num armazém.

Facilmente se conclui que a ordenação de dados surge com alguma frequência como um problema a resolver em programação. É normal que o programador recorra a funções/processos já implementados para efetuar a ordenação pretendida, não tendo portanto que implementar um algoritmo de ordenação. Todavia, é objetivo desta ficha exercitar o aluno na implementação de alguns algoritmos simples de ordenação.

Quando usar ordenação de dados?

É difícil estar a indicar todas as situações em que se deve ou não utilizar a ordenação de dados. Pode-se no entanto utilizar a seguinte regra orientadora: um conjunto de dados deve estar ordenado sempre que as tarefas executadas sobre ele se tornem consideravelmente mais fáceis/rápidas de executar pelo simples facto de os dados estarem ordenados.

Imagine que a base de dados em papel do registo civil dos cidadãos portugueses (a base de dados dos cartões de cidadão) não estava ordenada pelo número de identificação. Se fosse pedido a um funcionário para pesquisar fichas de cidadãos portugueses pelo seu número de identificação, ele teria que percorrer em média, e por cada pesquisa, metade das fichas de todos os portugueses (5 milhões de fichas). Isto seria obviamente inviável! Contudo, se os ficheiros em papel estiverem ordenados pelo número de identificação, ele só teria que ir à secção onde está o número respetivo e ler os dados de uma única ficha.

A base de dados dos cartões de cidadão tem também que estar ordenada por nome: imagine que se pede a um funcionário que procure a ficha de um português com um determinado nome, sem se conhecer o seu número de identificação. Se a base de dados não estiver ordenada também pelo nome, ele teria que ver as fichas de quase todos os portugueses, comparando o nome até encontrar o pretendido. Se a base de dados estiver também ordenada pelo nome, então a tarefa será bastante mais fácil.

A conclusão a tirar deste exemplo é que a existência de dados ordenados permite, na grande maioria das vezes, pesquisas muito mais rápidas.

Que algoritmo de ordenação devo usar?

Conhece-se atualmente muitos algoritmos de ordenação. Uns são muito rápidos a ordenar, mas bastante complexos, outros são mais lentos, mas mais simples de implementar e de entender.

Um programador que necessite de ordenar os seus dados recorre habitualmente a algoritmos já implementados ou que sabe implementar rapidamente. Ele só precisa de recorrer aos algoritmos mais eficientes e complexos se estiver a lidar com conjuntos de dados muito grandes, algures entre centenas de milhar a milhões de dados. Para ordenar conjuntos de dados de menor dimensão, um dos algoritmos mais simples cumpre quase de certeza a tarefa de ordenação de forma eficiente. O próprio C tem definida a função `qsort()` que implementa um algoritmo de ordenação bastante eficiente: o *quick sort*. De seguida, são apresentados alguns dos algoritmos de ordenação mais simples.

4.3 Ordenação por seleção (*selection sort*)

4.3.1 O algoritmo de seleção

Um dos algoritmos mais simples para ordenar um conjunto de dados é o algoritmo de seleção (*selection sort*). O princípio de funcionamento do algoritmo é o seguinte:

Imagine-se que se quer colocar por ordem crescente uma tabela de números inteiros. Percorre-se a tabela do início ao fim para encontrar o menor dos números, depois troca-se esse número com o existente na 1ª posição. Volta-se a repetir este processo, mas agora só a partir da 2ª posição. Depois o menor elemento encontrado é trocado com o elemento na 2ª posição. Repetindo este procedimento partindo sempre de uma posição mais à frente que a anterior, ficaremos no final do processo com a tabela ordenada por ordem crescente.

Vejamos um exemplo:

tab =	98	42	41	**23**	37	58
índice	0	1	2	**3**	4	5

Percorrendo a tabela do índice 0 ao 5 verificamos que é no índice 3 que se encontra o menor número da tabela. Fazemos pois uma troca entre o número no índice 0 (primeiro) e o número no índice 3.

tab =	23	42	41	98	**37**	58
índice	0	1	2	3	**4**	5

(A cinzento encontram-se os números ordenados e a branco os que ainda estão por ordenar)

Percorrendo a tabela do índice 1 ao 5, selecionamos o menor dos números, neste caso na posição 4. Trocamos pois os elementos nas posições 1 e 4.

tab =	23	37	**41**	98	42	58
Índice	0	1	**2**	3	4	5

Percorrendo do índice 2 ao 5, selecionamos o menor dos números, neste caso na posição 2. Nesta situação o número até já se encontra na posição correta.

tab =	23	37	41	98	**42**	58
Índice	0	1	2	3	**4**	5

Percorrendo do índice 3 ao 5, selecionamos o menor dos números, neste caso na posição 4. Trocamos pois os elementos nas posições 3 e 4.

tab =	23	37	41	42	98	**58**
índice	0	1	2	3	4	**5**

Percorrendo do índice 4 ao 5, selecionamos o menor dos números, neste caso na posição 5. Trocamos pois os elementos nas posições 4 e 5.

tab =	23	37	41	42	58	98
índice	0	1	2	3	4	5

Neste ponto, podemos concluir que o elemento na posição 5 é o maior de todos e que a tabela já se encontra ordenada.

Algumas notas:

- Este algoritmo pode ser implementado quer para ordenar por ordem crescente (menor número no início) quer por ordem decrescente (maior elemento no início).

- O algoritmo também pode ser implementado partindo do fim para o início, retendo os maiores valores no fim (supondo uma ordenação crescente).

4.3.2 Implementação em C/C++ do algoritmo de seleção

No programa abaixo, mostra-se uma possível implementação do algoritmo de seleção para ordenar uma tabela de inteiros por ordem crescente.

```
#include <iostream>
using namespace std;

const int N=6;   // número de elementos na tabela

//-----------------------------------------------------------------------------
int indiceMin(int tab[ ],int inicio,int fim) // Devolve o índice do menor elemento da
{                                            // Tabela entre os índices inicio e fim
  int i, idxMenor;
```

```
  idxMenor=inicio;                // Assume que o primeiro elemento é o menor
  for(i=inicio+1; i<=fim; i++)    // Percorre a tabela entre inicio+1 e fim
     if(tab[idxMenor]>tab[i])     // Se o elemento atual é menor do que aquele que
        idxMenor=i;               // considerávamos menor então o elemento actual passa
                                  // a ser considerado o menor.
  return(idxMenor);
}
//-----------------------------------------------------------------------------------
// Troca o elemento com índice i pelo elemento com índice j na tabela tab
void troca(int tab[ ], int i, int j) {
  int aux = tab[i];
  tab[i] = tab[j];
  tab[j] = aux;
}
//-----------------------------------------------------------------------------------
// Ordena uma tabela tab com comp elementos
void ordenacaoSelecao(int tab[ ],int comp) {
  for (int i = 0; i < comp-1; i++)
    troca(tab,i,indiceMin(tab,i,comp-1));  // Troca o elemento actual com o menor
}                                          // elemento dos restantes valores da tabela
//-----------------------------------------------------------------------------------
// Imprime os primeiros numElem elementos da tabela
void imprimeTabela(const char *nomeTab, int tab[], int numElem) {
  cout << nomeTab << " = ";        // Imprime nome da tabela =
  for (int i=0; i<numElem; i++) {  // percorre todos os elementos da tabela
    cout << tab[i] << " ";
  }
  cout << endl;   // muda de linha no final
}
//-----------------------------------------------------------------------------------
int main() {
  // Inicializa a tabela como constante
  int tabela[N] = {98, 42, 41, 23, 37, 58};

  imprimeTabela("Tabela original", tabela, N);
  ordenacaoSelecao(tabela, N);
  imprimeTabela("Tabela ordenada", tabela, N);
}
```

No programa anterior a função de ordenação `ordenacaoSelecao()` recorre a duas funções auxiliares: a primeira para encontrar o menor valor na tabela de uma dada posição até ao fim da tabela e a segunda para trocar dois elementos da tabela. Foi também criada uma função auxiliar `imprimeTabela()` que permite visualizar os elementos da tabela, antes e depois da tabela ser ordenada.

Note que as alterações a fazer à função `ordenacaoSelecao()` para que esta ordene por ordem decrescente, em vez de o fazer por ordem crescente, são mínimas. Como exercício, adapte esta função para permitir a ordenação por ordem decrescente.

4.4 Ordenação por bolha (*Bubble sort*)

4.4.1 O algoritmo de ordenação por bolha

O algoritmo de ordenação por bolha (*bubble-sort*) é um outro algoritmo simples de ordenação que consiste em "deslocar" repetidas vezes ao longo da tabela uma "bolha ordenadora de 2 elementos". A bolha corresponde a dois elementos sucessivos de uma tabela. Verifica-se se esses dois elementos já estão ou não na ordem certa. Se não estiverem, então são trocados. Se este procedimento for repetido um número de vezes suficiente, a tabela fica ordenada.

Voltando à tabela do exemplo anterior, coloca-se uma bolha ordenadora nos primeiros 2 elementos:

tab =	98	42	41	23	37	58
índice	0	1	2	3	4	5

Uma vez que estes dois elementos estão desordenados (42 devia estar antes do 98), são trocados. A bolha desloca-se de seguida para a direita. O resultado após a troca é:

tab =	42	98	41	23	37	58
índice	0	1	2	3	4	5

O processo repete-se: neste caso o 41 é menor do que o 98. Tem que ocorrer mais uma troca dentro da bolha para a tabela ficar ordenada.

tab =	42	41	98	23	37	58
índice	0	1	2	3	4	5

O procedimento é repetido até a bolha chegar ao final da tabela.

tab =	42	41	23	37	58	**98**
índice	0	1	2	3	4	5

Como se pode observar, a tabela ficou com o maior elemento na posição correta, ou seja na última posição da tabela. De seguida, volta-se a colocar a bolha no início da tabela.

tab =	42	41	23	37	58	**98**
índice	0	1	2	3	4	5

Repete-se este procedimento até que a bolha chegue ao penúltimo elemento (o último elemento já está ordenado).

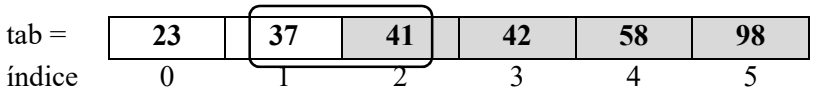
tab =	41	23	37	42	**58**	**98**
Índice	0	1	2	3	4	5

De seguida, volta-se a colocar a bolha no início e faz-se percorrer a tabela até ao antepenúltimo elemento:

tab =	23	37	41	**42**	**58**	**98**
índice	0	1	2	3	4	5

Neste momento, tem-se a certeza que, seja qual for a tabela, o número na posição 3 já está na posição correta. Neste exemplo em concreto, a tabela até já está toda ordenada, mas como o algoritmo tem de ser genérico e dar o resultado pretendido seja qual for a tabela, o processo tem que continuar.

Volta-se portanto a fazer percorrer a bolha ao longo da parte desordenada da tabela.



Quando a bolha de ordenação chega ao fim, sabe-se que o número na posição 2 também já está ordenado. Mais, sabe-se que, uma vez que não ocorreu qualquer troca de elementos, então a tabela já está ordenada, pois todos os pares consecutivos da tabela encontram-se na posição certa. Esta é uma particularidade do algoritmo de ordenação por bolha que o pode tornar ligeiramente mais rápido que o algoritmo de seleção.

Algumas notas:

- O algoritmo de ordenação por bolha pode ser implementado, quer para ordenar por ordem crescente (menor número no início) quer por ordem decrescente (maior número no início).

- O algoritmo pode também ser utilizado deslocando a "bolha" do fim para o início. Nesse caso, a bolha deixa de parar no primeiro valor da tabela para parar sucessivamente numa posição mais à frente.

4.4.2 Implementação em C/C++ do algoritmo de ordenação por bolha

No programa abaixo, apresenta-se uma possível implementação do algoritmo de ordenação por bolha para ordenar uma tabela do tipo `double` por ordem crescente. Mais uma vez, recorre-se a uma função auxiliar `troca()` que executa a tarefa de trocar dois elementos caso estes estejam desordenados.

```
#include <iostream>
using namespace std;

const int N=6;   // número de elementos na tabela
//-----------------------------------------------------------------------------
// Troca o elemento com índice i pelo elemento com índice j na tabela tab
void troca(double tab[ ], int i, int j) {
  double aux = tab[i];
  tab[i] = tab[j];
  tab[j] = aux;
}
//-----------------------------------------------------------------------------
void ordenacaoBolha(double tab[],int comp) {
  bool houveTroca;

  for (int i = 0; i < comp-1; i++) {
    houveTroca = false;
    for (int j = 1; j<comp-i; j++)
      if(tab[j]<tab[j-1]) {
        troca( tab, j-1, j);
        houveTroca = true;
      }
    if (!houveTroca) break;
  }
}
//-----------------------------------------------------------------------------
int main() {
  // inicializa a tabela como constante
  double tabela[N] = {9.8, 4.2, 4.1, 2.3, 3.7, 5.8};

  imprimeTabela("Tabela original", tabela, N);
  ordenacaoBolha (tabela, N);
  imprimeTabela("Tabela ordenada", tabela, N);
}
//-----------------------------------------------------------------------------
```

Mais uma vez, as alterações a fazer à função `ordenacaoBolha()` para que esta ordene por ordem decrescente em vez de o fazer por ordem crescente são mínimas (na realidade basta alterar a condição que determina a troca dos valores).

A variável `houveTroca` foi acrescentada para tornar o algoritmo mais rápido e permitir parar a ordenação a partir do momento que se saiba que a tabela se encontra ordenada (não houve trocas na interação anterior). Caso se retire todas as referências a esta variável o algoritmo funciona à mesma.

4.5 Ordenação por inserção (*insertion sort*)

4.5.1 O algoritmo de ordenação por inserção

O algoritmo de ordenação por inserção (*insertion sort*) é outro algoritmo simples de ordenação que consiste em ir ordenando os elementos de uma tabela como se faz com um baralho de cartas: inserindo na ordem correta os elementos a partir do segundo (o primeiro elemento está naturalmente ordenado). Considera-se que a tabela está dividida em duas partes. A parte esquerda está ordenada (começa com um único elemento, o primeiro elemento da tabela). A parte direita está desordenada e vai-se inserindo, ordenadamente, o primeiro elemento da parte desordenada na parte ordenada. Este processo é repetido até a parte desordenada desaparecer. Neste processo iterativo é usada uma variável temporária que guarda o elemento a inserir enquanto os elementos da parte esquerda (a já ordenada) vão sendo deslocados para a direita até se chegar à posição de inserção, inserindo-se aí o elemento guardado na variável temporária. Dos algoritmos de complexidade quadrática, este é o mais eficiente. Quando se pretende inserir um novo elemento numa tabela já ordenada, deve ser usado um algoritmo baseado neste.

Voltando à tabela do exemplo anterior, começa-se por considerar a parte já ordenada (a cinzento) apenas com o primeiro elemento da tabela, estando os restantes elementos desordenados. A seguir copia-se o elemento a inserir, o de índice 1, na variável temporária `temp`.

Tab =	98	**42**	41	23	37	58
índice	0	**1**	2	3	4	5

`temp` = 42

Percorrendo a tabela desde o índice 1 até ao índice 0, vai-se deslocando os elementos para a direita até encontrarmos um elemento menor que o elemento a inserir. Neste caso, só um elemento é deslocado e chega-se ao índice 0, ficando:

	↓					
Tab =	98	**98**	41	23	37	58
índice	0	**1**	2	3	4	5

`temp` = 42

Copia-se o valor da variável temporária para a tabela e prossegue-se da mesma forma. Agora a tabela já tem dois elementos ordenados e o novo elemento a inserir (o 3º) foi copiado para a variável temporária:

Tab =	42	98	**41**	23	37	58
índice	0	1	**2**	3	4	5

`temp` = 41

Percorrendo agora a tabela desde o índice 2 até ao índice 0, vai-se deslocando os elementos para a direita até encontrarmos um elemento menor que o a inserir. Neste caso dois elementos são deslocados e chega-se ao índice 0, ficando:

			↓			
Tab =	42	42	**98**	23	37	58
índice	0	1	**2**	3	4	5

temp = 41

Depois de se inserir ordenadamente o 3ºelemento da tabela e quando se vai inserir o 4º elemento, fica:

Tab =	41	42	98	**23**	37	58
índice	0	1	2	**3**	4	5

temp = 23

Depois de se deslocarem os elementos para a direita, fica:

		↓				
Tab =	41	41	42	**98**	37	58
índice	0	1	2	**3**	4	5

temp = 23

Depois de se inserir o 4ºelemento da tabela e quando se vai inserir o 5º elemento, temos:

Tab =	23	41	42	98	**37**	58
índice	0	1	2	3	**4**	5

temp = 37

Depois de se deslocarem os elementos para a direita, fica:

		↓				
Tab =	23	41	41	42	**98**	58
índice	0	1	2	**3**	4	5

temp = 37

Depois de se inserir o 5ºelemento da tabela e quando se vai inserir o 6º elemento, temos:

Tab =	23	37	41	42	98	**58**
índice	0	1	2	3	4	**5**

temp = 58

Depois de se deslocarem os elementos para a direita, fica:

					↓	
Tab =	23	37	41	42	98	**98**
índice	0	1	2	3	4	**5**

temp = 58

Finalmente, depois de se inserir o último elemento, obtemos a tabela ordenada:

Tab =	23	37	41	42	58	98
índice	0	1	2	3	4	5

4.5.2 Implementação em C/C++ do algoritmo de inserção

No programa abaixo, mostra-se uma possível implementação do algoritmo de inserção para ordenar uma tabela de inteiros por ordem crescente.

```
#include <iostream>
using namespace std;

const int N=6;   // número de elementos na tabela
//-----------------------------------------------------------------------------
// ordena por inserção vetor v com n elementos
void ordenacaoInsercao(int v[], int n) {
   for (int i = 1; i < n; i++) {  // n-1 iterações
      int aux = v[i];            // elemento a inserir copiado para var temporaria
      int j;                     // vai conter índice onde inserir
      for (j = i; (j > 0) && (aux < v[j-1]); j--)  // desloca elementos para direita
         v[j] = v[j-1];                            // ate' chegar à posição de insercao
      v[j] = aux;                                 // insere elemento no índice j
   }
}
//-----------------------------------------------------------------------------
// Imprime os primeiros numElem elementos da tabela
void imprimeTabela(const char *nomeTab, int tab[], int numElem) {
  cout << nomeTab << " = ";        // Imprime nome da tabela =
  for (int i=0; i<numElem; i++) {  // percorre todos os elementos da tabela
    cout << tab[i] << " ";
  }
  cout << endl;   // muda de linha no final
}
//-----------------------------------------------------------------------------
int main() {
  // inicializa a tabela como constante
  double tabela[N] = {9.8, 4.2, 4.1, 2.3, 3.7, 5.8};

  imprimeTabela("Tabela original", tabela, N);
  ordenacaoInsercao(tabela, N);
  imprimeTabela("Tabela ordenada", tabela, N);
}
//-----------------------------------------------------------------------------
```

Mais uma vez, as alterações a fazer à função `ordenacaoInsercao()` para que esta ordene por ordem decrescente em vez de o fazer por ordem crescente são mínimas (na realidade basta alterar a condição que determina a deslocação dos elementos).

4.6 Como usar a função qsort (ordenação «rápida» ou *quick sort*)

O C possui uma função de ordenação denominada `qsort()` definida na biblioteca `<cstdlib>`. Esta função implementa o algoritmo de ordenação «rápida», conhecido por *Quick Sort* e tem a capacidade de ordenar dados de diferentes tipos. Para tornar esta função genérica, é necessário fornecer uma função que permita comparar o tipo de dados a ordenar. A definição da função é a seguinte:

```
void qsort(void *base, size_t nelem, size_t width,
           int (*fcmp)(const void *, const void *));
```

`base` = ponteiro para o 1º elemento da tabela;

`nelem` = número de elementos dessa tabela;

`width` = número de bytes ocupado por cada elemento (use o `sizeof()` para o ajudar nesta tarefa);

`fcmp` = função de comparação entre dois elementos. Esta função tem que ser implementada pelo programador de acordo com o tipo de dados a utilizar como critério de ordenação. Deve devolver um número menor do que zero se o primeiro elemento for menor do que o segundo, zero se os dois elementos forem iguais, e um número maior do que zero se o primeiro elemento for maior que o segundo.

Vejamos o potencial da função `qsort()`**:** Imagine que se pretende ordenar uma tabela de fichas que contêm dois dados relativos a uma pessoa: nome e idade. Pretende-se ordenar esta tabela pelos nomes das pessoas. Para conseguir este propósito, poderemos utilizar a função `qsort()` da seguinte forma:

```
#include <iostream>   // cout
#include <iomanip>    // setw()
#include <cstring>    // strcmp()
#include <cstdlib>    // qsort()
using namespace std;
//-----------------------------------------------------------------------------
const int N=6;   // número de elementos na tabela
//-----------------------------------------------------------------------------
struct TFicha {  // definição de 1 ficha
  int idade;     // idade de cada pessoa
  char nome[21]; // nome da pessoa (até 20 caracteres + '\0')
};
//-----------------------------------------------------------------------------
int comparaNome(const void *ficha1, const void *ficha2) { // compara 2 nomes da ficha
  return strcmp( ((TFicha *)ficha1)->nome, ((TFicha *)ficha2)->nome);
  // usou-se a função strcmp que devolve um resultado igual ao pretendido
}
//-----------------------------------------------------------------------------
void imprimeTabela(const char *nomeTab, TFicha tab[], int numElem) {
  cout << nomeTab << " = " << endl;  // Escreve nomeTab =
  for (int i=0; i < numElem; i++) {  // percorre todos os elementos da tabela
    cout << "Nome: "<< setw(20)<< tab[i].nome << "   Idade: " << tab[i].idade << endl;
  }
}
//-----------------------------------------------------------------------------
int main() {
  // inicia-se a tabela no momento da definição
  TFicha tabela[N] = {35, "Rui" , 21, "Manuel", 20, "Jorge",
                      16, "Luis", 19, "Pedro" , 25, "Artur"};

  imprimeTabela("Tabela original", tabela, N);
  qsort((void *)tabela, N, sizeof(TFicha), comparaNome);
  imprimeTabela("Tabela ordenada", tabela, N);
}
//-----------------------------------------------------------------------------
```

Notas:

- O uso de `qsort()` obriga à inclusão da biblioteca `<cstdlib>` ou `<search.h>`;
- Quando se pretende armazenar nomes com 20 caracteres, por exemplo, deve-se somar mais um caráter para a terminação da *string*: o caráter `'\0'` (caráter com código ASCII 0);
- A função `compara_nome()` usa alguns *typecasts* (adaptação de tipos) necessários a uma correta utilização do `qsort()`. Na função `compara_nome()`, usou-se uma função da biblioteca `<cstring>` do C++, a função `strcmp()`, que tem um comportamento igual ao desejado para a função de comparação do

`qsort()` – devolve zero se as duas *strings* são iguais, um valor menor do que zero se a primeira *string* é menor do que a segunda, e um valor maior do que zero se a primeira *string* for maior do que a segunda;
- Esta secção pretendeu mostrar como fazer a ordenação de uma tabela sem haver a necessidade de implementar explicitamente um dos algoritmos estudados na secção anterior.

4.7 Geração de números aleatórios

Muitas vezes existe a necessidade de se recorrer à geração de números aleatórios, para os mais variados fins: nos jogos de computador estes são utilizados muitas vezes para criar um comportamento imprevisível para o jogador; nos simuladores para dar um comportamento mais próximo da realidade variável dos sistemas reais, etc.

Nesta ficha, a geração de números aleatórios é útil para testar os algoritmos de ordenação. Em vez de se definir explicitamente uma tabela de números inteiros, podemos recorrer a uma função geradora de números aleatórios para preencher uma tabela com números aleatórios. Imagine que se pretende preencher uma tabela `tab[]` de 20 inteiros, com números entre 0 e 100, escolhidos ao acaso. O código a seguir apresentado permite-nos precisamente isso:

```
for (int i=0; i<20; i++) {  // percorre os 20 elementos da tabela
  tab[i] = rand() % 101;    // atribui um valor ao "calhas" entre 0 e 100
}
```

A função `rand()` devolve um número inteiro aleatório compreendido entre `0` e `RAND_MAX` (constante definida em `<cstdlib>`). Para se gerar um inteiro compreendido entre 0 e 100, basta utilizar o módulo da divisão inteira (`%`). O resto da divisão de um número qualquer por 101 será sempre um número entre 0 e 100.

Imagine agora que se pretende preencher a mesma tabela `tab[]` de 20 inteiros escolhidos aleatoriamente, mas com números entre `valor_inicial` e `valor_final`. O código poderia ser:

```
for (int i=0; i<20; i++) {  // percorre os 20 elementos da tabela
  tab[i] = valor_inicial + rand() % ( valor_final - valor_inicial + 1 );
}   // atribui um valor ao "calhas" entre valor_inicial e valor_final (inclusive)
```

Para gerar números reais com uma casa decimal podemos gerar inteiros correspondendo às décimas desses números reais e depois fazer a divisão real por 10.0. No nosso caso ficaria:

```
// atribui um valor real ao "calhas",
// entre valor_inicial/10 e valor_final/10, inclusive, com "precisão" às décimas
   tab[i] = ( valor_inicial + rand() % ( valor_final – valor_inicial + 1) ) / 10.0;
```

Se experimentar um destes códigos, facilmente constatará que a sequência de números aleatórios gerada é sempre a mesma. Isto deve-se ao facto da função `rand()` basear a geração dos números aleatórios na utilização de um determinado número inteiro (designado por *semente*) como base para a sequência aleatória. Se esse número for sempre o mesmo, então a sequência de números aleatórios também será sempre a mesma. Para alterar este tipo de comportamento existe a função `srand(int seed)` que permite definir a semente a utilizar pelo `rand()`. Se alteramos o valor da semente a cada utilização do programa, garantimos que a sequência de números gerados vai ser sempre diferente. A função `time()` (definida em `time.h`) que devolve o número de segundos desde a meia-noite do dia 1 de Janeiro de 1970, pode ser utilizada para garantir que a semente dos números aleatórios é sempre diferente a cada execução do

programa. O código que permite inicializar a semente com valores diferentes a cada execução do programa será o seguinte:

```
#include <time.h>

...
srand((unsigned)time(NULL));
...
```

Sugere-se uma pesquisa às funções `srand()` e `rand()` para aprofundar os seus conhecimentos sobre geração de números aleatórios.

4.8 Ordenar um ficheiro de texto

Até agora apenas se considerou a ordenação de dados numa tabela em memória: uma tabela de inteiros, uma tabela do tipo `double` ou uma tabela de fichas. Em aplicações reais, os dados estão armazenados em ficheiros, o que determina que por vezes as operações de ordenação tenham que ser efetuadas em ficheiros.

A ordenação de um ficheiro de texto pode ser feita recorrendo temporariamente a uma tabela de *strings* em memória. Um procedimento simples para ordenar um ficheiro de texto pode resumir-se ao seguinte:

- Ler todo o ficheiro de texto para uma tabela de *strings* (cada *string* corresponde a uma linha);
- Aplicar um qualquer algoritmo de ordenação à tabela de *strings* em memória;
- Escrever a tabela de *strings* já ordenada para um ficheiro de texto em disco.

4.9 Ordenar um ficheiro binário

Um esquema simples de ordenação de um ficheiro binário obedece aos mesmos três passos apresentados na secção anterior: ler ficheiro, ordenar em memória, gravar para ficheiro. Ao contrário de um ficheiro de texto, um ficheiro binário pode conter uma enorme complexidade de dados. Um ficheiro binário tanto pode guardar os tipos mais usuais do C/C++, como `int`, `float` ou `double`, como pode guardar fichas/estruturas mais ou menos complexas.

Esta secção apresenta um exemplo de um programa para ordenar um ficheiro de fichas contendo dados sobre pessoas (nome e altura). Pretende-se reordenar o ficheiro de modo a que ele passe a estar ordenado pela altura das pessoas. São definidas duas funções auxiliares: para iniciar de forma aleatória o ficheiro e para mostrar os dados armazenados em ficheiro.

```
// Este programa demonstra a forma como ordenar um ficheiro binário contendo fichas

#include <iostream>   // cout, cin
#include <iomanip>    // setw(), setprecision()
#include <cstdio>     // EOF
#include <cstring>    // strcmp(), strcpy(), strcat()
#include <search.h>   // qsort()
#include <cstdlib>    // random()
#include <fstream>    // open(), write(), read(), close()
#include <time.h>     // time()
using namespace std;
//---------------------------------------------------------------------------------
const int NumFichas=22;  // número de fichas que o ficheiro possui
const char nomeFicheiro[] = "Dados.dat"; // nome do ficheiro binário
const char nomes[6][50] = { "Artur", "Bruno", "Carlos",   // nomes para preencher
                            "Diogo", "Eduardo", "Felix"}; // ficheiro
//---------------------------------------------------------------------------------
typedef struct{   // definição de 1 ficha contendo nome e altura de uma pessoa
  char nome[21];  // nome da pessoa (até 20 caracteres + '\0')
  double altura;  // altura de cada pessoa
}TRec;
//---------------------------------------------------------------------------------
// criaFicheiro
//---------------------------------------------------------------------------------
// Esta função cria um ficheiro de fichas do tipo TRec com n fichas geradas
// aleatoriamente. Devolve 1 se o ficheiro foi inicializado sem problemas, ou 0 se
// ocorreram problemas durante a geração do ficheiro.
//
int criaFicheiro(int n) {
  fstream fic; // variável que se vai associar ao ficheiro a abrir
  char c;      // variável auxiliar
  TRec aux;    // variável auxiliar

  cout << endl << endl << endl;
  fic.open(nomeFicheiro, ios::in); // verifica se ficheiro já existe
  if (fic.is_open()) { // aparentemente o ficheiro já existe. será que o utilizador
                       // quer escrever por cima?
    cout << "O ficheiro: " << nomeFicheiro << " já existe. Quer reescrever?" << endl;
    cout << "[S-SIM, qualquer outra tecla = NÃO]" << endl;
    c = cin.get();
    cin.ignore(); // limpa lixo do buffer de leitura
    if((c != 'S') && (c!='s')) { // não quer reescrever o ficheiro existente
      fic.close();              // fecha o ficheiro que acabou de abrir
      return 0;        // termina função sem ter criado um novo ficheiro aleatório
    }
    // chegou aqui porque o utilizador carregou na tecla S de SIM.
    fic.close();  // fecha o ficheiro que abriu para leitura
  }
  // a função continua na pág. seguinte
```

```
  // Se chegou aqui é porque vamos mesmo criar um novo ficheiro com dados aleatórios
  // o ficheiro a abrir é binário, para escrita e apaga o conteúdo anterior.
  fic.open(nomeFicheiro, ios::out | ios::trunc | ios::binary);
  if (!fic.is_open()) {
    cout << "Houve problemas a criar o ficheiro!!" << endl << "Prima uma tecla"<<endl;
    cin.get();   // espera que o utilizador prima uma tecla
    return 0;
  }
  for (int i = 0; i < n; i++) {
    strcpy( aux.nome, nomes[rand() % 6] );  // Gera um nome aleatório
    strcat( aux.nome, " ");                 // Acrescenta um espaço
    strcat( aux.nome, nomes[rand() % 6] );  // Acrescenta outro nome próprio
    aux.altura = (rand() % 80)/100.0+1.30;  // gera altura de 1.30 m a 2.10 m
    fic.write((char *)&aux, sizeof(aux));   // grava ficha no ficheiro
  }
  fic.close(); // fecha o ficheiro
  return 1;   // chegou aqui, o que quer dizer que tudo correu bem
} // fim de criaFicheiro
//-----------------------------------------------------------------------------
void imprimeFicheiro(void) {
  fstream fic;  // variável que vai ser associada ao ficheiro a ler
  int n;        // variável auxiliar para contar o número de fichas do ficheiro
  TRec aux;

  cout << endl << endl << endl;
  fic.open(nomeFicheiro, ios::in | ios::binary);
  if (!fic.is_open()) {
    cout << "houve problemas a abrir o ficheiro "<< nomeFicheiro << " para leitura\n";
    cout << "Prima qualquer tecla";
    cin.get();
    return;
  }
  // chegou aqui porque o ficheiro abriu corretamente
  n = 0;  // vai ler ficha na posição 0 do ficheiro
  while (fic.peek() !=.EOF) {  // le ficheiro enquanto houver fichas para ler
    fic.read( (char *)&aux, sizeof(aux) );
    cout << "(" << setw(2) << n << ") = " << setw(20) << aux.nome;
    cout << ", " << setprecision(2) << aux.altura << endl;
    n++;
  }
  fic.close();  // termina leitura do ficheiro
  cout << "Prima qualquer tecla" << endl;
  cin.get();
}
//-----------------------------------------------------------------------------

// o programa continua na pág. seguinte
```

```
//--------------------------------------------------------------------------------
// Função para comparar as alturas de 2 fichas (para o qsort)
//
int comparaAltura(const void *ficha1, const void *ficha2) {
  return ((TRec *)ficha1)->altura < ((TRec *)ficha2)->altura ? -1 : 1;
  // devolve -1 se altura 1 for menor que a altura 2, senão devolve 1
}
//--------------------------------------------------------------------------------
void ordenaFicheiro(void) {
  fstream fic;
  TRec *tab;

  cout << endl << endl << endl;
  fic.open(nomeFicheiro, ios::binary | ios::in );
  if (!fic.is_open()) {
    cout<<"Houve problemas a abrir o ficheiro "<< nomeFicheiro<<" para leitura"<<endl;
    cout << "Prima qualquer tecla para continuar" << endl;
    cin.get();
    return;
  }

  tab = new TRec[NumFichas];  // Aloca memória para a tabela
  fic.read((char *)tab, NumFichas * sizeof(TRec)); // Lê o ficheiro todo para memória
  fic.close();  // termina a leitura do ficheiro

  qsort((char *)tab, NumFichas, sizeof(TRec), comparaAltura); // ordena tabela

  fic.open(nomeFicheiro, ios::binary | ios::out | ios:: trunc);
  fic.write((char *)tab, NumFichas * sizeof(TRec)); // escreve tabela toda no ficheiro
  fic.close(); // termina a escrita no ficheiro

  delete [] tab;  // liberta toda a memória pedida para a tabela

  cout << "Ficheiro ordenado com sucesso..." << endl << endl;
}
//--------------------------------------------------------------------------------
int main() {
  char c; // variável auxiliar para opção do utilizador

  srand((unsigned)time(NULL));
  do {
    cout << "\n\n MENU PRINCIPAL:\n";
    cout << "1 - Gerar um novo ficheiro com dados aleatórios\n";
    cout << "2 - Ver conteúdo do ficheiro\n";
    cout << "3 - Ordenar ficheiro\n";
    cout << "4 - Sair do programa\n";
    cin.get(c);     // le opção
    cin.ignore();   // ignora lixo que fica no buffer de leitura
    switch (c) {
      case '1':
        criaFicheiro(NumFichas);
      break;
      case '2':
        imprimeFicheiro();
      break;
      case '3':
        ordenaFicheiro();
      break;
    }
  } while(c != '4'); // não termina programa enquanto o utilizador não carregar em '4'
}
//--------------------------------------------------------------------------------
// fim de programa
//--------------------------------------------------------------------------------
```

Notas:
- A maior parte do código é apenas auxiliar. As únicas funções que fazem mesmo a ordenação do ficheiro são a `comparaAltura()` e a `ordenaFicheiro()`;
- Procure entender o código apresentado: a maior parte dele representa tarefas elementares de programação;
- Foi utilizada a função `qsort()` para ordenar os dados. Recomenda-se que tente alterar o código para utilizar um dos algoritmos de ordenação apresentados nas aulas teóricas.

4.10 Análise da complexidade de algoritmos – notação O-grande

Em Ciências da Computação, faz-se a análise da complexidade dos algoritmos para se ter uma ideia da ordem de grandeza do tempo de execução dos algoritmos para um determinado nº de dados a processar. Os algoritmos são então classificados usando a notação O-grande (*Big-O notation*). Ao analisar a complexidade de um algoritmo em termos de tempo de processamento, vai ter-se em conta o nº de vezes que o conjunto de instruções mais frequentes é executado, em função do nº de dados N a processar. Assim, um algoritmo em que só existe um ciclo (for, while ou do…while) com N iterações tem complexidade linear, sendo classificado como O(N). O cálculo do valor máximo de uma tabela é um exemplo de um algoritmo com complexidade linear. Se houver dois ciclos imbricados a complexidade é quadrática, sendo a classificação $O(N^2)$. Um exemplo de um algoritmo com complexidade quadrática é o algoritmo de ordenação por seleção. Uma pesquisa binária tem complexidade O(log N). Na análise da complexidade de algoritmos não se tem em conta constantes simples (não dependentes do nº de dados a processar). Assim, um algoritmo de determinação simultânea do maior e do menor valor de uma tabela também tem complexidade linear, O(N).

4.11 Exercícios sugeridos

Pretende-se com o conjunto de exercícios apresentados a seguir que o aluno consolide os seus conhecimentos relativos à ordenação de dados e complexidade de algoritmos. Para permitir uma melhor orientação do estudo, cada exercício foi classificado de acordo com o seu nível de dificuldade. Aconselha-se o aluno a tentar resolver em casa os exercícios não realizados durante as aulas práticas.

Problema 4.1 – Ordenação por bolha (*Bubble-sort*) de inteiros + `random()` – Fácil

Faça um programa que permita ordenar por ordem crescente uma tabela com 20 inteiros (idades de eleitores) usando o algoritmo de ordenação por bolha (*bubble-sort*). Inicie a tabela no início do programa de forma aleatória, com números no intervalo de 18 a 90. Implemente também uma função que imprima os números da tabela. Utilize esta função para imprimir a tabela antes e depois de ter sido ordenada.

Problema 4.2 – Ordenação por seleção (*Selection-sort*) de reais – Fácil

Implemente uma função que permita ordenar por ordem decrescente uma tabela de reais (tipo `float`) utilizando obrigatoriamente o algoritmo de ordenação por seleção (*selection-sort*). Teste a função num programa feito por si e com os valores contidos na tabela a serem inicializados como constantes (esses valores serão temperaturas em 6 cidades europeias). A função deverá receber como parâmetro o número de elementos que a tabela possui.

Problema 4.3 – Ordenação de uma tabela de estruturas – Fácil

Implemente uma função que permita ordenar por ordem alfabética (do nome) uma tabela com 10 fichas (defina uma nova estrutura de dados usando `struct`). Cada ficha contém o nome, a idade e o número de estudante de um aluno. Teste a função num programa feito por si e com valores na tabela inicializados como constantes (use valores verosímeis).

Problema 4.4 – `qsort` de *ints* – Fácil /Médio

Resolva o problema anterior utilizando a função `qsort()`, disponível no ANSI C. Ordene por ordem crescente da idade.

Problema 4.5 – Ordenação de um *array* de *strings* – Médio

Faça uma função que permita ordenar uma tabela de nomes de países (*strings*). A função deverá receber como parâmetro o ponteiro para a tabela e o número de países na tabela. Teste essa função com um programa feito por si. Inicialize a tabela explicitamente com nomes de teste não ordenados e crie uma nova função para imprimir a tabela. Utilize esta função para mostrar a tabela antes e depois de ser ordenada.

Problema 4.6 – Ordenação de um ficheiro de texto – Médio

Faça um programa que ordene por ordem decrescente de idades as pessoas listadas num ficheiro de texto. O programa deve ler os dados do ficheiro para memória, ordená-los e voltar a escrevê-los no disco. O formato de cada linha do ficheiro é: *data_de_nascimento nome*. Exemplo: `19880501 José Silva`.

Para verificar o resultado do programa utilize uma tabela de *strings* com pelo menos 100 caracteres. Assuma que o ficheiro não tem mais do que 100 linhas de texto. Crie um ficheiro de texto (editado à mão num qualquer processador de texto) com nome `dados_desord.txt` e grave o resultado da ordenação em `dados_ord.txt`.

Problema 4.7 – Ordenação de um ficheiro de texto – `qsort()` – Médio

Resolva o problema anterior usando a função `qsort()`, disponível no ANSI C. Se não resolveu o problema anterior, faça a ordenação de uma tabela de *strings* iniciada na declaração.

Problema 4.8 – Ordenação de um ficheiro de texto – `qsort()` – Difícil

Resolva o problema anterior usando a função `qsort()` mas peça ao utilizador mais um parâmetro: a partir de que coluna do texto é que quer começar a ordenar o texto. Veja o resultado no exemplo seguinte:

conteúdo do ficheiro `nomes.txt`:	ordenado a partir da coluna 1	ordenado a partir da coluna 10:
`19880521 Carlos`	`19781021 Artur`	`19781021 Artur`
`19781021 Artur`	`19880521 Carlos`	`20000501 Bruno`
`20000501 Bruno`	`20000501 Bruno`	`19880521 Carlos`

Problema 4.9 – Ordenação de um *array* de *structs* - Difícil

Faça um programa que ordene uma tabela com 12 fichas criadas por si explicitamente. Cada ficha deve ter 3 campos: uma *string* de 60 caracteres (nome), um número inteiro sem sinal (idade) e um `float` (altura em metros). Deverá iniciar a tabela usando uma função que coloque dados desordenados nas 12 fichas e em todos os campos. Deve depois mostrar um menu de opções para o utilizador com o seguinte aspeto:

```
MENU:
1 – Iniciar com os dados de arranque (desordenados) e mostrar no monitor
2 – Ordenar por nome e mostrar no monitor
3 – Ordenar pela idade e mostrar no monitor
4 – Ordenar pela altura e mostrar no monitor
5 – Sair do programa (sem confirmação)
```

Problema 4.10 – `qsort()` de uma tabela de `structs` - Difícil

Use a função `qsort()` para resolver o problema anterior.

Problema 4.11 – Q*uick-sort* versus *bubble-sort* - Difícil

Faça um programa que permita comparar o desempenho em termos de tempo de execução dos algoritmos *quick-sort* (usando a função `qsort()`, disponível no ANSI C) e *bubble-sort*. Esse programa deve ter 2 opções: testar *quick-sort* e testar *bubble-sort*. Para fazer cada um dos testes, deve fazer o seguinte: iniciar uma tabela com 100000 elementos do tipo `double` gerados aleatoriamente; depois, use a função `time()` da biblioteca `<time.h>` para guardar o tempo em segundos, antes de chamar o algoritmo de ordenação e depois deste terminar; calcular e mostrar no monitor a diferença entre os 2 instantes, em segundos.

Para efeitos de verificação, teste primeiro os dois algoritmos com uma tabela com apenas 10 elementos e imprima-os no monitor para confirmar que o programa está a funcionar bem. Calcule os tempos para a tabela com 100000 elementos do tipo `double` e para uma tabela com 200000 elementos do tipo `double`. Conclua como evolui o tempo de execução de cada um dos algoritmos quando aumenta a tabela para o dobro.

Problema 4.12 – *Selection sort* versus `qsort()` versus *bubble-sort* - Difícil

Altere o programa anterior de forma a permitir comparar o desempenho em termos de tempo de execução dos algoritmos *selection sort*, *quick-sort* (usando a função `qsort()` disponível no ANSI C) e *bubble-sort*. Esse programa deve ter 3 opções: testar *selection sort,* testar *quick-sort* e testar *bubble-sort*.

Para efeitos de verificação, teste primeiro os 3 algoritmos com uma tabela com apenas 10 elementos e imprima-os no monitor para confirmar que o programa está a funcionar bem. Calcule os tempos para a tabela com 100000 elementos do tipo `double` e para uma tabela com 200000 elementos do tipo `double`. Conclua como evolui o tempo de execução de cada um dos algoritmos quando aumenta o tamanho da tabela para o dobro.

Problema 4.13 – Ordenação de fichas com o Quick-Sort – Difícil (adaptado do teste de frequência de 16/03/2016)

Considere uma tabela de fichas (`struct`) com informação sobre ratos de computador à venda numa determinada loja. Cada ficha tem os seguintes campos:

- `sem_fios` (booleano; `true` se rato sem fios)
- `marca` (*string* à C, até 40 caracteres úteis)
- `ref` (referência; *string* à C, até 20 caracteres úteis)
- `em_stock` (indica quantos ratos com esta referência existem para venda na loja).

Considere que existe uma tabela de fichas de tipos de rato definida e inicializada na função `main()`.

a) Declare a estrutura e chame-lhe rato.

b) Escreva uma função para indicar quantos ratos com fios existem para venda. A função tem 2 parâmetros: uma tabela de fichas e um inteiro com o nº de fichas existentes na tabela. Vai retornar o nº de ratos com fios existentes.

c) Implemente uma função que permita ordenar uma tabela de fichas de ratos pela sua marca, usando o algoritmo Quick-Sort (`qsort`).

Nenhuma destas funções escreve nada no ecrã. Como trabalho extra, implemente uma função `main()` mínima para ir testando as funções implementadas.

Problema 4.14 – Inserção ordenada por *Bubble-sort* – Difícil (adaptado do teste de frequência de 25/03/2015)

Considere uma `struct` para gerir uma tabela ordenada de números reais (amostras de temperaturas). Os campos são a tabela propriamente dita (`tab`), a sua dimensão (`max`) e um inteiro (`n`) com o nº de elementos na tabela. A tabela deve estar sempre ordenada por ordem decrescente, com os dados agrupados no início da tabela.

a) Declare a `struct` e defina uma função para inicializar uma tabela vazia com dimensão dada pelo 2.º parâmetro dessa função. A `struct` da tabela a inicializar é o 1.º parâmetro da função.

b) Implemente uma função `insereTab()` que acrescenta um novo elemento, ordenadamente, a uma tabela, devolvendo `false` se a inserção falhar por a tabela estar cheia e `true` caso contrário. Os dois parâmetros da função são: a `struct` com a tabela e o novo elemento a inserir. Utilize o princípio do algoritmo de ordenação da bolha, assumindo que a tabela está ordenada antes da inserção do novo elemento.

Problema 4.15 – Ordenação por inserção – Difícil (adaptado do Exame de Recurso de 03/07/2017)

Considere uma tabela de fichas (`struct`) com informação sobre ratos de computador à venda numa determinada loja. Cada ficha tem os seguintes campos:

- `pvp` (preço em euros, real de precisão dupla)
- `marca` (*string* à C++)
- `ref` (referência, *string* à C, até 20 caracteres úteis)
- `stock` (indica quantos ratos destes existem para venda na loja).

a) Declare a estrutura e chame-lhe rato.

b) Implemente uma função que permita ordenar por ordem decrescente uma tabela de fichas de ratos pelo seu preço, usando o algoritmo de ordenação por inserção. Esta função não escreve nada no ecrã.

Problema 4.16 – Inserção ordenada – Difícil (adaptado do teste de frequência de 13/03/2018)

Considere uma tabela de fichas (tabela de `struct`) com informação sobre monitores de computador à venda numa dada loja. Cada ficha tem os seguintes campos:

. `marca` (*string* à C, até 50 caracteres úteis)
. `ref` (referência; *string* à C, até 30 caracteres úteis)
. `fullHD` (booleano, `true` se formato `fullHD`)
. `em_stock` (indica quantos monitores destes existem para venda na loja, máx. até cerca de 700).

a) Declare a estrutura e chame-lhe monitor.
b) Implemente uma função (devolve `void`) que permita inserir ordenadamente um novo tipo de monitor (ainda não existente) numa tabela de fichas de monitores, ordenada pela marca, usando o princípio do algoritmo da ordenação por inserção. Parta do princípio de que nunca se atinge a dimensão máxima da tabela. A função não escreve nada no ecrã.
c) Declare e inicialize na função `main()` uma tabela (ordenada pela marca) de tipos de monitores existentes e uma variável inteira com o nº de elementos existentes nessa tabela.
d) Chame a função da alínea b) para inserir ordenadamente uma nova ficha (novo tipo de monitor), declarada e inicializada na função `main()`.
e) Imprima a lista de marcas existentes.

Problema 4.17 – Complexidade de algoritmos – Fácil

Considere uma tabela com `N` números inteiros ordenada por ordem crescente. Indique exemplos de operações sobre essa tabela que tenham complexidade: a) constante; b) linear; c) logarítmica; d) quadrática.

Problema 4.18 – Complexidade de algoritmos – Fácil

Considere uma função `g()` de complexidade O(N) e outra função `f()` de complexidade $O(N^2)$ que quando manipulam `N` números reais demoram cerca de 10 ms cada uma a serem executadas. Se essas funções passarem a manipular 10×N números reais qual o tempo expectável de processamento?

Problema 4.19 – Complexidade de algoritmos – Fácil

Considere uma tabela com `N` nomes de pessoas (strings C++) ordenada por ordem alfabética. Usando a notação O-grande, indique a complexidade dos seguintes algoritmos:

a) ordenação da tabela (por ordem alfabética) usando um algoritmo da bolha standard;
b) ordenação da tabela (por ordem alfabética inversa) usando um algoritmo da bolha standard;
c) pesquisa sequencial de um nome;
d) pesquisa binária de um nome.